LEARN C++
Skill up with our free tutorials

2.7 — Forward declarations and definitions

ALEX[1] APRIL 20, 2024

Take a look at this seemingly innocent sample program:

```
 1  #include <iostream>
 2
 3  int main()
 4  {
 5      std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';
 6      return 0;
 7  }
 8
 9  int add(int x, int y)
10  {
11      return x + y;
12  }
```

You would expect this program to produce the result:

```
The sum of 3 and 4 is: 7
```

But in fact, it doesn't compile at all! Visual Studio produces the following compile error:

```
add.cpp(5) : error C3861: 'add': identifier not found
```

The reason this program doesn't compile is because the compiler compiles the contents of code files sequentially. When the compiler reaches the function call to *add* on line 5 of *main*, it doesn't know what *add* is, because we haven't defined *add* until line 9! That produces the error, *identifier not found*.

Older versions of Visual Studio would produce an additional error:

```
add.cpp(9) : error C2365: 'add'; : redefinition; previous definition was 'fo
```

This is somewhat misleading, given that *add* wasn't ever defined in the first place. Despite this, it's useful to generally note that it is fairly common for a single error to produce many redundant or related errors or warnings. It can sometimes be hard to tell whether any error or warning beyond the first is a consequence of the first issue, or whether it is an independent issue that needs to be resolved separately.

Best practice

When addressing compilation errors or warnings in your programs, resolve the first issue listed and then compile again.

To fix this problem, we need to address the fact that the compiler doesn't know what add is. There are two common ways to address the issue.

Option 1: Reorder the function definitions

One way to address the issue is to reorder the function definitions so *add* is defined before *main*:

```
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';
    return 0;
}
```

That way, by the time *main* calls *add*, the compiler will already know what *add* is. Because this is such a simple program, this change is relatively easy to do. However, in a larger program, it can be tedious trying to figure out which functions call which other functions (and in what order) so they can be declared sequentially.

Furthermore, this option is not always possible. Let's say we're writing a program that has two functions *A* and *B*. If function *A* calls function *B*, and function *B* calls function *A*, then there's no way to order the functions in a way that will make the compiler happy. If you define *A* first, the compiler will complain it doesn't know what *B* is. If you define *B* first, the compiler will complain that it doesn't know what *A* is.

Option 2: Use a forward declaration

We can also fix this by using a forward declaration.

A **forward declaration** allows us to tell the compiler about the existence of an identifier *before* actually defining the identifier.

In the case of functions, this allows us to tell the compiler about the existence of a function before we define the function's body. This way, when the compiler encounters a call to the function, it'll understand that we're making a function call, and can check to ensure we're calling the function correctly, even if it doesn't yet know how or where the function is defined.

To write a forward declaration for a function, we use a **function declaration** statement (also called a **function prototype**). The function declaration consists of the function's return type, name, and parameter types, terminated with a semicolon. The names of the parameters can be optionally included. The function body is not included in the declaration.

Here's a function declaration for the *add* function:

```
int add(int x, int y); // function declaration includes return type, name, parameters, and semicolon.  No function body!
```

Now, here's our original program that didn't compile, using a function declaration as a forward declaration for function *add*:

```
#include <iostream>

int add(int x, int y); // forward declaration of add() (using a function declaration)

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n'; // this works because we forward declared add() above
    return 0;
}

int add(int x, int y) // even though the body of add() isn't defined until here
{
    return x + y;
}
```

Now when the compiler reaches the call to *add* in main, it will know what *add* looks like (a function that takes two integer parameters and returns an integer), and it won't complain.

It is worth noting that function declarations do not need to specify the names of the parameters (as they are not considered to be part of the function declaration). In the above code, you can also forward declare your function like this:

```
int add(int, int); // valid function declaration
```

However, we prefer to name our parameters (using the same names as the actual function). This allows you to understand what the function parameters are just by looking at the declaration. For example, if you were to see the declaration `void doSomething(int, int, int)`, you may think you remember what each of the parameters represent, but you may also get it wrong.

Also many automated documentation generation tools will generate documentation from the content of header files, which is where declarations are often placed. We discuss header files and declarations in lesson 2.11 -- Header files (https://www.learncpp.com/cpp-tutorial/header-files/)[2].

Best practice

Keep the parameter names in your function declarations.

Tip

You can easily create function declarations by copy/pasting your function's header and adding a semicolon.

Why forward declarations?

You may be wondering why we would use a forward declaration if we could just reorder the functions to make our programs work.

Most often, forward declarations are used to tell the compiler about the existence of some function that has been defined in a different code file. Reordering isn't possible in this scenario because the caller and the callee are in completely different files! We'll discuss this in more detail in the next lesson (2.8 -- Programs with multiple code files (https://www.learncpp.com/cpp-tutorial/programs-with-multiple-code-files/)[3]).

Forward declarations can also be used to define our functions in an order-agnostic manner. This allows us to define functions in whatever order maximizes organization (e.g. by clustering related functions together) or reader understanding.

Less often, there are times when we have two functions that call each other. Reordering isn't possible in this case either, as there is no way to reorder the functions such that each is before the other. Forward declarations give us a way to resolve such circular dependencies.

Forgetting the function body

New programmers often wonder what happens if they forward declare a function but do not define it.

The answer is: it depends. If a forward declaration is made, but the function is never called, the program will compile and run fine. However, if a forward declaration is made and the function is called, but the program never defines the function, the program will compile okay, but the linker will complain that it can't resolve the function call.

Consider the following program:

```cpp
#include <iostream>

int add(int x, int y); // forward declaration of add()

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';
    return 0;
}

// note: No definition for function add
```

In this program, we forward declare *add*, and we call *add*, but we never define *add* anywhere. When we try and compile this program, Visual Studio produces the following message:

```
Compiling...
add.cpp
Linking...
add.obj : error LNK2001: unresolved external symbol "int __cdecl add(int,
add.exe : fatal error LNK1120: 1 unresolved externals
```

As you can see, the program compiled okay, but it failed at the link stage because *int add(int, int)* was never defined.

Other types of forward declarations

Forward declarations are most often used with functions. However, forward declarations can also be used with other identifiers in C++, such as variables and types. Variables and types have a different syntax for forward declaration, so we'll cover these in future lessons.

Declarations vs. definitions

In C++, you'll frequently hear the words "declaration" and "definition" used, and often interchangeably. What do they mean? You now have enough fundamental knowledge to understand the difference between the two.

A **declaration** tells the *compiler* about the *existence* of an identifier and its associated type information. Here are some examples of declarations:

```
int add(int x, int y); // tells the compiler about a function named "add" that takes two int parameters and returns an int.  No body!
int x;                 // tells the compiler about an integer variable named x
```

A **definition** is a declaration that actually implements (for functions and types) or instantiates (for variables) the identifier.

Here are some examples of definitions:

```
int add(int x, int y) // implements function add()
{
    int z{ x + y };   // instantiates variable z

    return z;
}

int x;                // instantiates variable x
```

In C++, all definitions are declarations. Therefore `int x;` is both a definition and a declaration.

Conversely, not all declarations are definitions. Declarations that aren't definitions are called **pure declarations**. Types of pure declarations include forward declarations for function, variables, and types.

Nomenclature

In common language, the term “declaration” is typically used to mean “a pure declaration”, and “definition” is used to mean “a definition that also serves as a declaration”. Thus, we’d typically call `int x;` a definition, even though it is both a definition and a declaration.

When the compiler encounters an identifier, it will check to ensure use of that identifier is valid (e.g. that the identifier is in scope, that it is used in a syntactically valid manner, etc…).

In most cases, a declaration is sufficient to allow the compiler to ensure an identifier is being used properly. For example, when the compiler encounters function call `add(5, 6)`, if it has already seen the declaration for `add(int, int)`, then it can validate that `add` is actually a function that takes two `int` parameters. It does not need to have actually seen the definition for function `add` (which may exist in some other file).

However, there are a few cases where the compiler must be able to see a full definition in order to use an identifier (such as for template definitions and type definitions, both of which we will discuss in future lessons).

Here’s a summary table:

Term	Technical Meaning	Examples

Declaration	Tells compiler about an identifier and its associated type information.	void foo(); // function forward declaration (no body)
		void goo() {}; // function definition (has body)
		int x; // variable definition

Definition	Implements a function or instantiates a variable. Definitions are also declarations.	void foo() { } // function definition (has body) int x; // variable definition
Pure declaration	A declaration that isn’t a definition.	void foo(); // function forward declaration (no body)
Initialization	Provides an initial value for a defined object.	int x { 2 }; // 2 is the initializer

The term “declaration” is commonly used to mean “pure declaration”, and the term “definition” used for anything that is both a definition and a declaration. We use this common nomenclature in the example column comments.

()The one definition rule (ODR) (#ODR)[4]

The **one definition rule** (or ODR for short) is a well-known rule in C++. The ODR has three parts:

1. Within a *file*, each function, variable, type, or template can only have one definition. Definitions occurring in different scopes (e.g. local variables defined inside different functions, or functions defined inside different namespaces) do not violate this rule.

2. Within a *program*, each function or variable can only have one definition. This rule exists because programs can have more than one file (we'll cover this in the next lesson). Functions and variables not visible to the linker are excluded from this rule (discussed further in lesson 7.6 -- Internal linkage[5]).
3. Types, templates, inline functions, and inline variables are allowed to have duplicate definitions in different files, so long as each definition is identical. We haven't covered what most of these things are yet, so don't worry about this for now -- we'll bring it back up when it's relevant.

Related content

We discuss ODR part 3 exemptions further in the following lessons:

- Types (13.1 -- Introduction to program-defined (user-defined) types[6]).
- Function templates (11.6 -- Function templates[7] and 11.7 -- Function template instantiation[8]).
- Inline functions and variables (5.7 -- Inline functions and variables[9]).

Violating part 1 of the ODR will cause the compiler to issue a redefinition error. Violating ODR part 2 will cause the linker to issue a redefinition error. Violating ODR part 3 will cause undefined behavior.

Here's an example of a violation of part 1:

```cpp
int add(int x, int y)
{
     return x + y;
}

int add(int x, int y) // violation of ODR, we've already defined function
add(int, int)
{
     return x + y;
}

int main()
{
    int x{};
    int x{ 5 }; // violation of ODR, we've already defined x
}
```

In this example, function `add(int, int)` is defined twice (in the global scope), and local variable `int x` is defined twice (in the scope of `main()`). The Visual Studio compiler thus issues the following compile errors:

```
project3.cpp(9): error C2084: function 'int add(int,int)' already has a bo
project3.cpp(3): note: see previous definition of 'add'
project3.cpp(16): error C2086: 'int x': redefinition
project3.cpp(15): note: see declaration of 'x'
```

However, it is not a violation of ODR part 1 for `main()` to have a local variable defined as `int x` and `add()` to also have a function parameter defined as `int x`. These definitions occur in different scopes (in the scope of each respective function), so they are considered to be separate definitions for two distinct objects, not a definition and redefinition of the same object.

For advanced readers

Functions that share an identifier but have different sets of parameters are also considered to be distinct functions, so such definitions do not violate the ODR. We discuss this further in lesson 11.1 -- Introduction to function overloading (https://www.learncpp.com/cpp-tutorial/introduction-to-function-overloading/)[10].

Quiz time

Question #1

What is a function prototype?

Hide Solution (javascript:void(0))[11]

> A function prototype is a declaration statement that includes a function's name, return type, parameter types, and optionally the parameter names. It does not include the function body. It tells the compiler about the existence of a function before it is defined.

Question #2

What is a forward declaration?

Hide Solution (javascript:void(0))[11]

> A forward declaration tells the compiler that an identifier exists before it is actually defined.

Question #3

How do we declare a forward declaration for functions?

Hide Solution (javascript:void(0))[11]

> For functions, a function declaration/prototype serves as a forward declaration.

Question #4

Write the function declaration for this function (use the preferred form with names):

```
int doMath(int first, int second, int third, int fourth)
{
     return first + second * third / fourth;
}
```

Show Solution (javascript:void(0))[11]

Question #5

For each of the following programs, state whether they fail to compile, fail to link, or compile and link successfully. If you are not sure, try compiling them!

a)

```
#include <iostream>
int add(int x, int y);

int main()
{
    std::cout << "3 + 4 + 5 = " << add(3, 4, 5) << '\n';
    return 0;
}

int add(int x, int y)
{
    return x + y;
}
```

Hide Solution (javascript:void(0))[11]

> Doesn't compile. The compiler will complain that it can't find a matching `add()` function that takes 3 arguments. The forward declaration of `add()` only has two parameters.

b)

```
#include <iostream>
int add(int x, int y);

int main()
{
    std::cout << "3 + 4 + 5 = " << add(3, 4, 5) << '\n';
    return 0;
}

int add(int x, int y, int z)
{
    return x + y + z;
}
```

Hide Solution (javascript:void(0))[11]

> Doesn't compile. The compiler will complain that it can't find a matching `add()` function that takes 3 arguments. The forward declaration of `add()` only has two parameters, and the definition of function `add()` that has 3 parameters hasn't been seen yet.

c)

```
#include <iostream>
int add(int x, int y);

int main()
{
    std::cout << "3 + 4 = " << add(3, 4) << '\n';
    return 0;
}

int add(int x, int y, int z)
{
    return x + y + z;
}
```

Hide Solution (javascript:void(0))[11]

> Doesn't link. The compiler will match the forward declaration of add to the function call to add() in main(). However, no add() function that takes two parameters was ever implemented (we only implemented one that took 3 parameters), so the linker will complain.

d)

```
#include <iostream>
int add(int x, int y, int z);

int main()
{
    std::cout << "3 + 4 + 5 = " << add(3, 4, 5) << '\n';
    return 0;
}

int add(int z, int y, int x) // names don't match the declaration
{
    return x + y + z;
}
```

Hide Solution (javascript:void(0))[11]

> Compiles and links. The types in the function call to add() matches the forward declaration, and the definition of add() also matches. The fact that the names don't match the declaration doesn't matter, as the names in a declaration are optional (and if provided, ignored by the compiler).

e)

```cpp
#include <iostream>
int add(int, int, int);

int main()
{
    std::cout << "3 + 4 + 5 = " << add(3, 4, 5) << '\n';
    return 0;
}

int add(int x, int y, int z)
{
    return x + y + z;
}
```

Hide Solution (javascript:void(0))[11]

> Compiles and links. This is the same as the prior case. Function declarations do not need to specify the names of the parameters (even though we generally prefer to include them).

Next lesson
2.8 Programs with multiple code files

3

Back to table of contents

12

Previous lesson
2.6 Why functions are useful, and how to use them effectively

13

14

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[16] are connected to your provided email address.

509 COMMENTS Newest

Archit Kumar
March 13, 2024 4:10 am

I have a question, what if there's a function header, forward function declaration but nothing in the function body. What will happen in this case? The program will compile but fail to link or won't compile at all?

0 Reply

> **Alex** Author
> Reply to Archit Kumar [17] March 13, 2024 10:26 pm
>
> By nothing in the function body, I assume you mean no body, not an empty body `{}`.
>
> It should compile, as the compiler will assume the function must be defined elsewhere. If the function is not called, the program will link. If the function is called, the linker will error.
>
> 1 Reply

Anon
March 6, 2024 8:21 pm

Why do we prefer to include the parameter names in the function declaration? If I change the parameter names in the function I don't want to forget to change them in the declaration too. A quick test on my system compiled when the parameter names of the declaration didn't match the names in the function itself. In practice this could cause a lot of confusion.

This compiles and works, for example:

```cpp
#include <iostream>

int add(int a, int b); // function declaration with two parameters named a
and b

int main()
{
    std::cout << add(1, 2) << '\n';
    return 0;
}

int add(int x, int y) // note that the function parameter names don't match
the function declaration parameter names
{
    return x + y;
}
```

Last edited 1 month ago by Anon

0 Reply

Alex Author

Reply to Anon 18 March 9, 2024 4:53 pm

Putting names in your declarations helps prevent usage mistakes. If you see some function declaration `void doSomething(int, int, int)`, you may think you remember what each of the `int` parameters mean... but you may also be wrong.

Also many automatic documentation generating tools generate the documentation from the content of header files, which is where the declarations of functions defined in source files live.

Changing parameter names isn't particularly common, and when they do change, they probably won't change much. You might change `name` to `studentName` or something, but if the definition uses `studentName` and the declaration uses `name` this is unlikely to cause much confusion.

3 Reply

JPerk

Reply to Anon 18 March 9, 2024 12:52 am

!DSICLAIMER : I could be wrong, i am learning aswell.!

Perhaps it works because once the compiler knows that it exists, when it comes back to check the add function, it just redifines it ? like it just goes through the line and changes (int a, int b) to (int x, int y)

idk. just letting my thoughts out.

0 Reply

Anon
March 4, 2024 7:40 am

May I suggest swapping the order of Declaration and Definition in the summary table, so they appear in the same order in both text and the table?

0 Reply

Alex Author
Reply to Anon 19 March 5, 2024 2:07 pm

Done.

0 Reply

IceFloe
February 20, 2024 6:26 am

Thanks for the lesson, I wanted to clarify the question, isn't the declaration part of the definition, well, I mean, the definition is the totality of the declaration? And initialization is when we set the initial value of a variable?

0 Reply

Alex Author
Reply to IceFloe 20 February 22, 2024 1:37 pm

All definitions are declarations. There are declarations that are not definitions. So one is not strictly equivalent to the other in all cases.

Initialization is setting the initial value when the variable is actually instantiated.

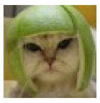
0 Reply

Luis
February 3, 2024 12:28 am

Are these considered declarations and/or definitions?

```
int x = 5;
int x {5};
```

0 Reply



Alex Author

Reply to Luis [21] February 3, 2024 11:46 am

Conventionally we'd say these are definitions. However, all definitions are declarations, so technically they are both.

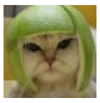
0 Reply

Nulllegz

January 26, 2024 4:10 pm

Hiya, for question 5c, I understand this program fails to link since there's no function that's implemented with the same number of parameters within this source file. However, I was wondering, since a program can have multiple source files, if there was a defined function add() with 2 parameters in a different source file, could this program run correctly with the given forward declaration at the top of this program?

0 Reply



Alex Author

Reply to Nulllegz [22] January 28, 2024 4:35 pm

Yep!

1 Reply

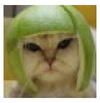
Swaminathan

January 19, 2024 5:44 am

Hello Alex, can you add which lessons discusses these? (understand it might be a complex topic for beginners, but you can add a note “For advanced readers”)

“Types, templates, inline functions, and inline variables are allowed to have duplicate definitions in different files, so long as each definition is identical. We haven’t covered what most of these things are yet, so don’t worry about this for now -- we’ll bring it back up when it’s relevant.”

0 Reply



Alex Author

Reply to Swaminathan [23] January 19, 2024 9:50 am

Added.

0 Reply

Swaminathan
January 19, 2024 3:10 am

Hello Alex, would you suggest any books/references to understand generally the "thought processes" & principles of a compiler & linker? By knowing those (like how compiler treats objects, variables, comments and its general order of working), I think I can get a deeper understanding of programming concepts which are roughly just expressions of those principles.

Last edited 3 months ago by Swaminathan

0 Reply

> **Alex** Author
> Reply to Swaminathan 24 January 19, 2024 9:36 am
>
> I don't have any to suggest in this area. Sorry.
>
> 0 Reply

Swaminathan
January 18, 2024 11:44 pm

Hello Alex, "*or* reader understanding" or "*for* reader understanding"?

0 Reply

> **Alex** Author
> Reply to Swaminathan 25 January 19, 2024 9:32 am
>
> The "or" is intentional here, as maximizing organization may not maximize comprehension.
>
> 0 Reply

tiro
December 25, 2023 9:45 am

Hello,

I happen to be a bit confused about how the executable changes with forward declarations.

"**All of the scanning is done at compilation time. When your compiler turns your code into an executable, everything is optimized, and function calls are replaced with jumps to particular addresses.
**"
Does it mean that forward declarations are only needed for the compiler not to display an error but the code would run fine otherwise?

Do forward declaration affect the executable or the efficency of the code in any way? (compared to just writing the definition before the call)

P.S. : Thank you for this website Alex.

0 Reply

Alex Author
Reply to tiro 26 December 28, 2023 12:36 pm

Yes, forward declarations are just for the compiler's sake, so it can perform proper and consistent type checking. They do not impact the executable or efficiency of the code in any way.

1 Reply

Lucas A
Reply to tiro 26 December 28, 2023 10:54 am

Because C++ is a compiled language the code cannot run without being compiled and therefore the compiler error essentially means the code does not run fine. Picture the code almost like its a book, reading from top to bottom, if you define a new function after the main() function it won't know what you're talking about because it hasn't read it yet, but the function declaration tells the compiler/program itself "Hey, there exists a function named foo() in this code, go find it so you aren't confused later"

2 Reply

Ahmed Mahmoud
December 5, 2023 10:53 pm

Hey Alex, I believe you should include this example in the questions section, along with this quote as a reminder:
It is worth noting that function declarations do not need to specify the names of the parameters (as they are not considered to be part of the function declaration):

```
#include <iostream>
int add(int x, int y, int z);

int main()
{
    std::cout << "3 + 4 + 5 = " << add(3, 4, 5) << '\n';
    return 0;
}

int add(int z, int y, int x)
{
    return (x*9) + y + z;
}
```

Last edited 4 months ago by Ahmed Mahmoud

0 Reply

Alex Author

Reply to Ahmed Mahmoud [27] December 7, 2023 12:57 pm

Added. Thanks for the suggestion!

2 Reply

Akshay

October 27, 2023 5:59 am

Hi Alex,

Firstly, I would like to thank you for creating and maintaining such a good tutorial, it has provided tons of Knowledge and Insights of C++.

I would like to recommend on correction in this page i.e., in ODR Section it is mentioned that, "Within a given file, a function, variable, type, or template can only have one definition." and "Within a given program, a variable or normal function can only have one definition.", please mention here that these rules only applies when the variables are declared in the Same Scope OR Global Scope or if it is visible to the linker during linking process, because it is a bit confusing for new readers if they consider local variables.

Thanks!!

Last edited 5 months ago by Akshay

1 Reply

Alex Author

Reply to Akshay [28] October 27, 2023 1:46 pm

Added. Thanks for the suggestion.

0 Reply

Akshay
Reply to Alex 29 October 27, 2023 11:36 pm

Thanks!

0 Reply

Serif
September 30, 2023 4:30 am

what is the difference between instantiate and initialize in variables?

0 Reply

Alex Author
Reply to Serif 30 October 2, 2023 9:46 am

Instantiation is allocating memory for the variable. Initialization is giving the variable an initial value upon creation.

6 Reply

Steve
September 22, 2023 1:35 pm

I can't quite wrap my head around the reason why the compile error in **Question #5 a)** is different from **Question #5 b)**.
If the compiler finds an issue at line 6 due to a discrepancy between the number of arguments in the function call and the forward declared function at line 2, why do the actual function differences in line 10 and 12 even matter? Shouldn't the compiler throw an error and stop before even checking the actual function definition? Maybe I'm misunderstanding how exactly the compiler works.

Also a minor point of confusion about the wording used in the answer to 5a) that refers to "same number of parameters". We learned before that the values passed from a caller to a function are called arguments. It seems to me that arguments are called parameters here. Can they be used interchangeably or did I misunderstand something.

Last edited 7 months ago by Steve

0 Reply

Alex Author

Reply to Steve [31] September 26, 2023 3:55 pm

5a and 5b should fail identically. I updated the text to make this more clear and fix the arguments/parameters terminology issue.

1 Reply

Gamborg

September 14, 2023 1:23 pm

Where the chain jumps off the bike a bit for me, is the understanding that a computer reads from above and goes downwards. It does not know anything below from where it is currently working. So how does it know what is within the add function, which is located after the main body? It has not reached that far down yet to see it? Or does it stop and look through everything when the add() function is called?

In short, what exactly is the computer doing once it reaches line 6 and encounteres the add() function. It has not read the body of the add() function which is located on lines 11-13?

```
 1  #include <iostream>
 2  int add(int x, int y, int z);
 3
 4  int main()
 5  {
 6      std::cout << "3 + 4 + 5 = " << add(3, 4, 5) << '\n';
 7      return 0;
 8  }
 9
10  int add(int x, int y, int z)
11  {
12      return x + y + z;
13  }
```

4 Reply

Andius pandius

Reply to Gamborg [32] April 4, 2024 10:11 am

Gamborg,
So understand the difference between compiling code and executing code. When the *compiler* gets to line 6, all it needs to know at that point is what the add function looks like to the outside world, ie what does it need (parameters) and what comes back (the result type). At this point it does not need to know what the add function does. The *compiler* is not *executing* your code.

Once the code is compiled and you are executing it then yes, the CPU does definitely need to know what add does at that point in the execution. But now the compiler and linker have put together the assembly for the CPU to know where to go find the definition for add.

0 Reply

Gamborg

Reply to Gamborg 32 September 14, 2023 1:38 pm

I am asuming this is the answer, given by admin Alex back in 2022 to user Arjan

All of the scanning is done at compilation time. When your compiler turns your code into an executable, everything is optimized, and function calls are replaced with jumps to particular addresses.

Ill leave the comment be for the next reader that might sit with the same question. Unless you feel it is redundant, then feel free to remove it.

8 Reply

lostlevy

August 25, 2023 10:38 am

```cpp
#include <iostream>
// This program is designed to mimic a simple calulator which perform add,
subtract, mul and div.

int add(int x, int y);          // Adds two numbers.
int mul(int x, int y);          // Multiply two numbers.
float divi(float x, float y);   // Divide two numbers.
int sub(int x, int y);          // Subtract two numbers.
int value1();                   // Ask user for first value.
int value2();                   // Ask user for second value.


int main()
{
    //Addition calculator
    std::cout << "Numbers to add\n";
    int a{ add(value2(), value1()) };
    std::cout << "Addition is " << a << '\n';
    std::cout << '\n';

    //Subtraction calculator
    std::cout << "Numbers to subtract\n";
    int b{ sub(value2(), value1()) };
    std::cout << "Subtraction is " << b << '\n';
    std::cout << '\n';

    //Division calculator
    std::cout << "Numbers to divide\n";
    float c{ divi(value2(), value1()) };
    std::cout << "Division is " << c << '\n';
    std::cout << '\n';

    //Multiplication calculator
    std::cout << "Numbers to multiple\n";
    int d{ mul(value2(), value1()) };
    std::cout << "Multiplication is " << d << '\n';
    std::cout << '\n';

    return 0;

}

/* Improvements Could have used print function.
                Could have used if else function for better designed calculator.
                Need to copy and past many things like adding a new line and defining a variable to print the solution.
                Overall calculator has a bad user experience in terms of
choosing what operation to first.
                Division dont produce decimals value after a certain point.
*/
```

```
int add(int x, int y)
{
    return x + y;
}

int mul(int x, int y)
{
    return x * y;
}

float divi(float x, float y)
{
    return y / x;
}

int sub(int x, int y)
{
    return y - x;
}


int value1()
{
    std::cout << "First number: ";
    int x{};
    std::cin >> x;
    return x;
}

int value2()
{
    std::cout << "Second number: ";
    int x{};
    std::cin >> x;
    return x;
}
My calculator
```

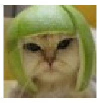
3 Reply

Krishnakumar
August 24, 2023 5:32 am

>We'll discuss this in more detail in the **next** lesson

This wording will break if/when lessons are reorganised. For consistency with all other lessons thus far, perhaps best to point to the relevant lesson's link and leave it at that (this is the style adopted in this tutorial series thus far for any cross-references).

1 Reply



Alex Author

Reply to Krishnakumar 33 August 28, 2023 1:03 pm

I'm okay with this. I think it's useful for readers who are following the tutorial series sequentially to know that this topic will be covered subsequently rather than in some far-away lesson.

5 Reply

Krishnakumar

Reply to Alex 34 November 15, 2023 6:25 am

Hmm alright, but it is slightly inconsistent with other lessons where immediately preceding or succeeding lessons are explicitly linked without the use of "previous" or "next" wording.

0 Reply

Krishnakumar

Reply to Alex 34 October 10, 2023 10:19 am

In technical writing, what we learnt is that it is better to cross-reference to exact lesson and have the renderer automatically convert that to next/previous wording or point to the actual hyperlink if it is further than that. This helps in easy reorganisation in the future. If the cross-references lesson(s) gets moved, then hard-coding the next/previous etc wording will break, right?

See for example the details in:

- https://en.wikibooks.org/wiki/LaTeX/Labels_and_Cross-referencing
- https://www.overleaf.com/learn/latex/Cross_referencing_sections%2C_equations_and_floats

0 Reply

Undead34

Reply to Krishnakumar 33 August 26, 2023 2:41 pm

I don't think it's necessary because this is supposed to be viewed sequentially and also you have to think that maybe at the time he wrote this he hadn't written the other section, I honestly don't understand why you are writing such an unnecessary comment.

5 Reply

Krishnakumar

Reply to Undead34 35 October 10, 2023 10:36 am

>I honestly don't understand why you are writing such an unnecessary comment.

I offer an explanation in my comment above. To summarize, it was based on the principles of cross-referencing in standard technical/academic writing.

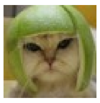
0 Reply

Olga

August 15, 2023 10:38 pm

Hey Alex! For variable Definition: isn't it type + identifier + assign a value? As opposed to variable declaration.

0 Reply



Alex Author

Reply to Olga 36 August 17, 2023 6:11 pm

No, a definition does not require an initializer (except in certain cases).

0 Reply

Undead34

Reply to Alex 37 August 26, 2023 2:41 pm

Ah those certain cases are so difficult

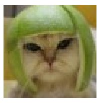
1 Reply

N T

August 5, 2023 10:28 pm

Is violation of part 2 of the ODR also technically undefined behaviour, since it is undetectable by the compiler?

0 Reply



Alex Author

Reply to N T 38 August 6, 2023 10:23 pm

No. Calling a function that has a forward declaration but no definition isn't detectable by the compiler, but it's not undefined behavior -- it's an error. So being undetectable by the compiler does not mean undefined behavior.

That said, there are cases where violating ODR part 2 can result in undefined behavior (e.g. an inline function with non-duplicate definitions) instead of a linker error.

0 Reply

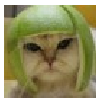
Krishnakumar

Reply to Alex [39] October 10, 2023 10:20 am

>That said, there are cases where violating ODR part 2 can result in undefined behavior (e.g. an inline function with non-duplicate definitions) instead of a linker error.

Is an example available somewhere later in this tutorial series?

0 Reply



Alex Author

Reply to Krishnakumar [40] October 11, 2023 3:24 pm

Sorry, an inline function with non-duplicate definitions would be a violation of ODR part 3 (not 2). My error.

0 Reply

Krishnakumar

Reply to Alex [41] November 15, 2023 6:27 am

No worries. The updated wordings on ODR violation aspects summarised in this chapter is really useful.

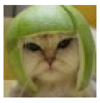
0 Reply

N T

August 5, 2023 5:28 pm

In the section of `Declaration vs. definitions`, would it possible to also add `int x` to the code snippet for definitions (after "Here are some examples of definitions:")? Currently, it is only shown as an example of a declaration (in the code snippet after "Here are some examples of declarations:"), which might give a reader who has not read '1.3 - Introduction to objects and variables' the impression that `int x;` is only a declaration.

0 Reply



Alex Author

Reply to N T [42] August 6, 2023 9:42 pm

Done.

0 Reply

Harsha Varthan
July 1, 2023 3:27 am

Can a function be declared multiple times in a translation unit?

```
int add(int, int);
int add(int, int);
```

```
int main(){
std::cout << add(3, 4) << '\n';
return 0;
}
```

```
 int add(int x, int y){
return x+y;
}
```

0 Reply

Alex Author
Reply to Harsha Varthan [43] July 3, 2023 3:07 pm

Declared, yes. Defined, no.

2 Reply

Harsha Varthan
June 22, 2023 10:54 am

does declaration of a variable or a function allocates memory in c++?

1 Reply

Alex Author
Reply to Harsha Varthan [44] June 23, 2023 10:52 pm

A declaration does not allocate memory. A definition does.

3 Reply

Harsha Varthan
June 22, 2023 10:32 am

"Forward declarations can also be used to define our functions in an order-agnostic manner. This allows us to define functions in whatever order maximizes organization (e.g. by clustering related functions together) or reader understanding."

This means Forward declarations can be useful for organizing code in a way that maximizes readability and understanding. For example, we could forward declare all of the functions in a module at the beginning of the module, and then define them in any order we want. This would allow us to cluster related functions together, and it would also make the code easier to read and understand because the user would not have to jump around to different parts of the code to find the definitions of the functions.

0 Reply

samuel
June 9, 2023 10:49 am

whatsa linker?

0 Reply

> **oe1243**
> Reply to samuel [45] June 11, 2023 4:25 pm
>
> what we call compilation process is actually countains 4 steps preprocessor, compiler, assembler and linker. after that each deferent file of the program has been converted to 0s and 1s the linker links those resulted files into one executable file
>
> *Last edited 10 months ago by oe1243*
>
> 4 Reply

Tom
June 9, 2023 6:20 am

is it important to differentiate whether compiler, linker or anything else fails? i can clearly see that in question 5 A, B and C they were written incorrectly and program would not work as i wanted.

Last edited 10 months ago by Tom

0 Reply

Alex Author

Reply to Tom 46 June 12, 2023 10:47 am

Sometimes knowing that the linker is failing rather than the compiler can help you diagnose what the issue is (e.g. something not having the correct linkage, or a file not being added to the project correctly).

2 Reply

Krishnakumar

May 25, 2023 5:04 pm

>However, forward declarations can also be used with other identifiers in C++, such as variables and **user-defined types**. Variables and **user-defined** types have a different syntax ....

Program-defined types?

Last edited 10 months ago by Krishnakumar

0 Reply

Alex Author

Reply to Krishnakumar 47 May 29, 2023 9:23 pm

Removed "user-defined" and just went with "types". That feels simpler to me and avoids using terms that haven't been defined yet.

0 Reply

Krishnakumar

May 25, 2023 5:00 pm

>However, we prefer to name our parameters (using the same names as the actual function), because it allows you to understand what the function parameters are just by looking at the declaration. Otherwise, you'll have to locate the function definition.

A major drawback of this approach is that, if the function definition (possibly in some other file) is changed to use some different variables, the variables in the declaration become out of sync with those in the definition, which becomes confusing.

Let's say we are calculating the simple interest and the parameter list of the function api could be `(int n, double r, double p)`. In the future, if we decide that we shall probably move the rate to the last, perhaps to facilitate default a default value (say a fixed default rate of interest), then if we had hard-coded the declaration, this is semantically misleading and potentially dangerous. However, if we had used only the type names in the function prototype, they will still remain valid.

0 Reply

Swaminathan
Reply to Krishnakumar 48 January 19, 2024 2:32 am

It seems exponentially more dangerous to switch parameter meanings which “will” impact potentially 100s(if are lucky) of users who had already used it basing previous documentation. Example: simple_interest(9, 7.5, 10000). This now calculates simple interest for 9 years, and 7.5 USD principal with 10000% interest?

0 Reply

Alex Author
Reply to Krishnakumar 48 May 29, 2023 9:20 pm

True in theory, but practically speaking I think this is almost a non-issue. We rarely reorder function parameters, and since doing so breaks the interface of the function, if we do then we should be extra careful to check ALL declarations, definitions, and calls to that function to make sure they are compliant with the new ordering.

On the other hand, having access to a declaration is common. And if you see `foo(int, int, int)`, you have no information about what the parameters mean. That seems like a more likely vector for errors to be made to me.

1 Reply

Krishnakumar
Reply to Alex 49 August 24, 2023 6:21 am

Hmm. I am 50-50 on this. But happy to accept this consensus and move on.

0 Reply

DeadlyAhmed
April 20, 2023 6:23 pm

what does instantiated mean ?

0 Reply

Alex Author
Reply to DeadlyAhmed 50 April 23, 2023 3:57 pm

With objects, instantiated means an object is created. Objects are sometimes called instances, and instantiated means creating an instance.

0 Reply

Blue Stew
March 20, 2023 6:33 pm

I think you touched on this in a separate comment, but I'd like more detail if possible...

So when the compiler generates the object code, does it treat each function as its own entity? I thought the linker only need to connect objects that are external to a particular file, so if you wrote a program in a single file, the linker would only need to link functions (for example) in the header files, standard library, etc. But your explanation of how the compiler and linker work and what will make each step fail seem to indicate that even the functions that are defined within the same file need to be linked if one calls another. I know the compiler doesn't create an object file for each function, but does it treat each function as a separate object which then need to be linked?

1 Reply

Alex Author
Reply to Blue Stew [51] March 23, 2023 1:46 pm

The compiler should be able to connect uses of an identifier with the definition for the identifier when they are in the same compilation unit. The linker only sees objects with external linkage, so that it can connect uses of an identifier in one compilation unit to a definition of that identifier in another compilation unit (and detect ODR violation across multiple files).

I rewrote the declaration vs definition section to remove some of the wording around compiler vs linker.

0 Reply

Krishnakumar
Reply to Alex [52] May 25, 2023 5:21 pm

This is an important, yet subtle point. The compiler is the one that tries to connect up the things within the same TU. If I think about it, it is obvious, but if you hadn't mentioned this in this comment, I wouldn't have thought about this point (even though we can figure it out with some thinking). Perhaps better to expand on this a little bit here, and perhaps more so in the lesson on internal and external linkage.

0 Reply

Alex Author

Reply to Krishnakumar [53] May 29, 2023 10:22 pm

I added an insight box to the next lesson (2.8) with some information about this. It seemed more relevant to discuss after our first multi-file example.

0 Reply

Krishnakumar

Reply to Alex [54] August 24, 2023 6:26 am

Yes. Reads much better now. Thank you for the additional explanation.

0 Reply

Anthony Schwartz

January 30, 2023 2:30 am

I really appreciate the step by step approach.

I know you get a lot of comments on how to improve your tutorials, or what you have done wrong. For me this is the perfect tutorial as someone who needs this kind of structure.

Thank you :)

8 Reply

Swaminathan

Reply to Anthony Schwartz [55] January 19, 2024 12:03 am

This is rather made “more” perfect by comments and corrections.

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/header-files/
3. https://www.learncpp.com/cpp-tutorial/programs-with-multiple-code-files/
4. https://www.learncpp.com/cpp-tutorial/forward-declarations/#ODR
5. https://www.learncpp.com/cpp-tutorial/internal-linkage/

6. https://www.learncpp.com/cpp-tutorial/introduction-to-program-defined-user-defined-types/
7. https://www.learncpp.com/cpp-tutorial/function-templates/
8. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/
9. https://www.learncpp.com/cpp-tutorial/inline-functions-and-variables/
10. https://www.learncpp.com/cpp-tutorial/introduction-to-function-overloading/
11. javascript:void(0)
12. https://www.learncpp.com/
13. https://www.learncpp.com/cpp-tutorial/why-functions-are-useful-and-how-to-use-them-effectively/
14. https://www.learncpp.com/forward-declarations/
15. https://www.learncpp.com/wordpress/recent-news-box-for-tiga-102-theme-updated/
16. https://gravatar.com/
17. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-594621
18. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-594376
19. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-594249
20. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-593856
21. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-593182
22. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-592933
23. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-592564
24. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-592558
25. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-592553
26. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-591337
27. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-590586
28. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-589162
29. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-589169
30. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-587992
31. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-587639
32. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-587221
33. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-586063
34. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-586281
35. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-586192
36. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-585625
37. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-585731
38. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-585151
39. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-585212
40. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-588441
41. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-588566
42. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-585145
43. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-582931
44. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-582285
45. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-581530
46. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-581492
47. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-580792
48. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-580791
49. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-580962
50. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-579561
51. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-578559
52. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-578622

53. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-580795
54. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-580966
55. https://www.learncpp.com/cpp-tutorial/forward-declarations/#comment-576933